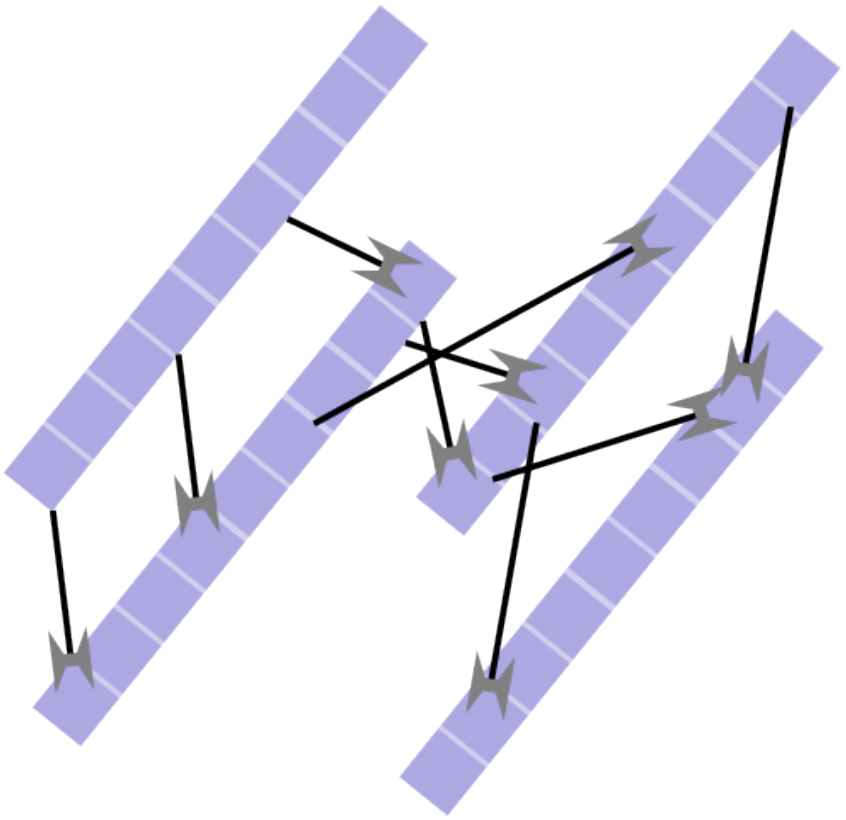
streamdice

Release 1.0

Andrew Garcia, Ph.D.

Apr 28, 2023

CONTENTS



A stream cipher developed by Andrew Garcia based on catalogued shuffled keyboards.

A cipher is an encryption algorithm and thus, can be applied to program development in any language. Originally developed in Python and optimized to C++ as **streamdice**. The JavaScript implementation **streamdiceJS** was written to integrate to websites as an interactive application.

Read the white paper: *Streamdice: An encryption algorithm based on catalogued shuffled keyboards*

Check out the *Usage* section for further information, including how to *streamdice.py* the project. For some quick examples / templates, check out the next section.

CHAPTER

ONE

CONTENTS

1.1 Usage

1.1.1 streamdice.py

Andrew Garcia, 2021

Oldest implementation of streamdice algorithm. Written in Python

- Encrypts all characters, including spaces.
- Requires 1 key as an input, but the root block and SEQUENCE block of the streamdice algorithm are preserved.

make app.sh executable with `chmod +x app.sh` and run

```
cd pystreamdice
chmod +x app.sh
./app.sh        // input command
```

1.1.2 streamdice ("streamdice++")

C++ implementation to streamdice

Usage

Silent Keys (with app.sh)

Command line in *streamdice/* directory:

make app.sh executable with *chmod +x app.sh*

Encryption

```
cd streamdice
./app.sh 1  // input command
```

```
>>> [OUT]
enter key #1:
enter key #2:
enter message:
Hello World. Hello again! olleH (5 letters) is 'Hello' backwards.
```

(continues on next page)

(continued from previous page)

```
--- message encrypted! ---
I4tY# 7#CYYC {2Y$S sv`yjG 1Y$BI Br nE&yW[ti ,v VI4tY#V ];~IrO.5vQ
```

Decryption

```
cd streamdice
./app.sh 0  // input command
```

```
>>> [OUT]
enter key #1:
enter key #2:
enter message:
I4tY# 7#CYYC {2Y$S sv`yjG 1Y$BI Br nE&yW[ti ,v VI4tY#V ];~IrO.5vQ
--- message encrypted! ---
Hello World. Hello again! olleH (5 letters) is 'Hello' backwards.
```

Explicit Keys (with direct ./streamdice)

Command line in *streamdice/build/* directory:

```
./streamdice 145 145236 1        // input command for encryption
enter message:

./streamdice 145 145236 0            // input command for decryption
enter message:
```

Details

```
// Input command format
./streamdice [ key1 ] [ key2 ] [ encrypt[1]/decrypt[0] ]  %

// Inputs
[ key1 ]    type long;   range:   0 to 2147483647  // "keep it shorter than 10 digits"
[ key2 ]    type long;   range:   1 to (2147483647 or < message_size)
[ encrypt ] type int;    range:   1 or 0 (True or False)

// Characters not supported:
\  ? | "
```

Usage password/generator.cpp

Klang it! Install klang (https://github.com/andrewrgarcia/klang)

```
cd password
klang generator.cpp
./generator.k
```

Installation Quick Start

A basic installation template.

Installing Boost

```
##### Linux

apt-get install -y libboost-iostreams-dev

##### macOS
brew install boost
##### Windows

vcpkg install boost-iostreams:x64-windows
vcpkg install boost-any:x64-windows
vcpkg install boost-algorithm:x64-windows
vcpkg install boost-uuid:x64-windows
vcpkg install boost-interprocess:x64-windows
```

Building StreamDice

```
git clone git@github.com: ...
cd streamdice/streamdice; mkdir build; cd build; cmake ..; cmake --build .
```

1.1.3 streamdiceJS

streamdiceJS is the interactive web [JavaScript] implementation of the StreamDice cipher.

Usage

web-ready

Check out the online demo

Local computer

```
cd streamdiceJS
npm run build
```

Then apply dist/main.js to web

1.2 API Reference

CHAPTER

TWO

WHITEPAPER

2.1 Streamdice: An encryption algorithm based on catalogued shuffled keyboards

Andrew R. Garcia

garcia.gtr@gmail.com

2.1.1 Abstract

ImageMesh is a method available in versions $\geq$ 2.0 of the voxelmap Python library that generates 3D models from images using Convex Hull in 3-D to enclose external points obtained from a series of partitioned point clouds. These point clouds are generated by assigning the relative pixel intensities from the partitioned images as the depth dimension to the points. In this paper, we describe the limitations of the original ImageMesh method and the quick solution we have implemented to address them. Additionally, we introduce MeshView, a Python visualization tool developed in tandem with ImageMesh that provides a convenient way to visualize the 3D models generated by ImageMesh. Finally, we discuss the GPU memory space complexity of both methods.

Good encryption can be used to protect data and private information. When properly encrypted, even if data is accessed in an unauthorized manner or unwillingly disclosed, the non-consented reader will be unable to read it without the correct encryption keys. The algorithm presented here, **streamdice**, is a stream cipher which encrypts characters (i.e. letters, numbers and some allowed signs) by both their specific identity as well as their relative location in the message thread. For streamdice, the stream units are shuffled keyboards generated by a pseudo-random number generator (PRNG), each of which are shifted once for every single encrypted character. The shuffled keyboards are limited and kept in memory with hashes, which are in turn dependent on the provided keys for encryption. The pseudo-random factor obfuscates the periodicity of the algorithm, and the encryption operations make it challenging to exploit by brute force.

2.1.2 Method

This method uses a hashmap where each one of its keys corresponds to an index of the keyboard character representing it in QWERTY order. For instance, the first keys for Q, W, and E characters are 1, 2 and 3, respectively. This ordered arrangement is analogous to a symbolic keyboard as the one in Figure 1.

For the encryption, the character values of the hashmap are shuffled with a pseudo-random number generator (PRNG) seeded by a "hash" number. For instance, shuffling the original keyboard in Figure 1 with a Mulberry32 PRNG and a #5443 hash will give the shuffled keyboard in Figure 2. Because the values are shuffled, the keys will be preserved, and thus the ciphertext can be transformed back by reference to the original keyboard, i.e.

$$H23(\text{ciphertext}) \rightarrow [4, 2, 13](\text{indices}) \rightarrow tea(\text{text})$$

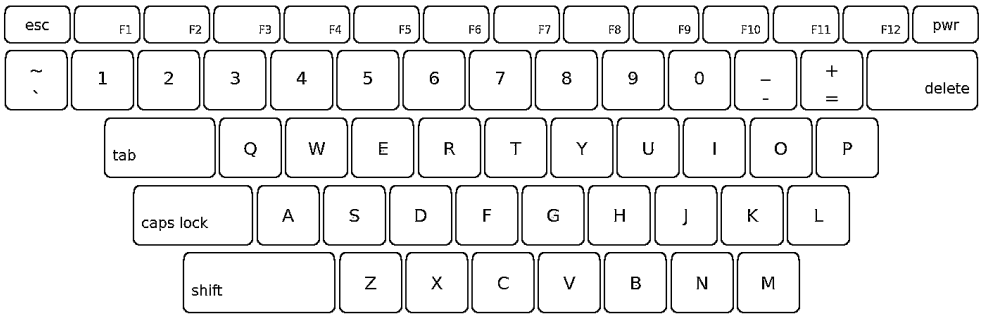


Fig. 1: Standard QWERTY keyboard

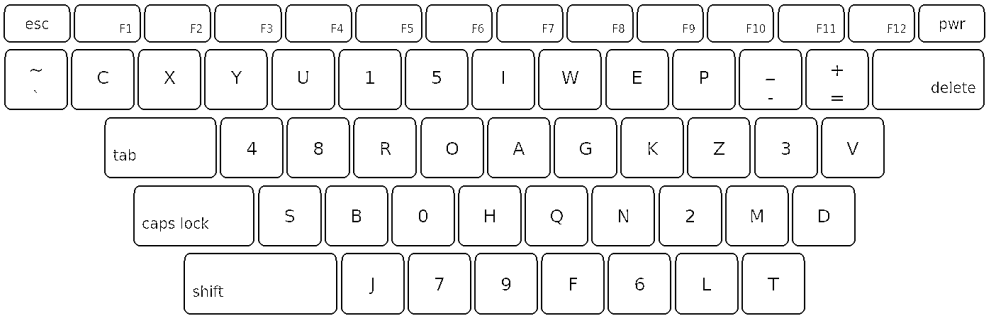


Fig. 2: Randomly-shuffled keyboard with #5443

The keyboard is shuffled each time a new character is encrypted with N permutations, calling a new shuffling operation by its PRNG hash as shown in Figure 3. As seen in the same figure, the permuted keyboards repeat periodically if the number of keyboards is less than the number of characters to encrypt. The specific hashes are computed directly from the 2 keys provided by the user for the encryption. An $\mathbf{H}$ vector contains all the $\mathrm{H_i}$ hashes used to generate the shuffled keyboards. The N number of $\mathrm{H_i}$ hashes is equal to the number of digits provided for $\mathrm{key_2}$ and are computed in the following way:

$$\Gamma_i = (\mathrm{key_2}//10^i)\%10$$
$$\mathrm{H_i} = \mathrm{key_1} + \Gamma_i$$

The hashes used to generate the new keyboard, rather than the specific keyboard arrangement, are the objects kept in memory throughout the encryption. As suggested above, the decryption takes the keys used to encrypt the messages and reverses the protocol. This method, thus, optimizes auxiliary space, $\mathcal{O}(N)$, rather than encryption time complexity, $\mathcal{O}(MC)$, where N is the number of digits of the key_2, while M and C are the message length and number of keyboard characters to encrypt, respectively.

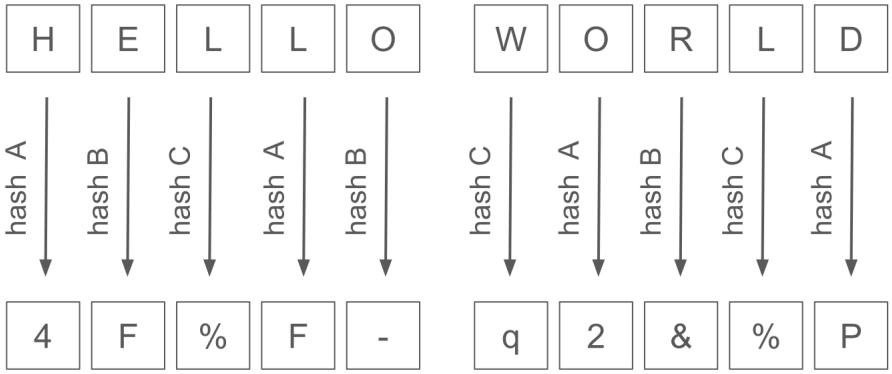


Fig. 3: Encrypting *hello world* with a periodically-repeating stream of 3 shuffled keyboards.